WL-TR-93-5025

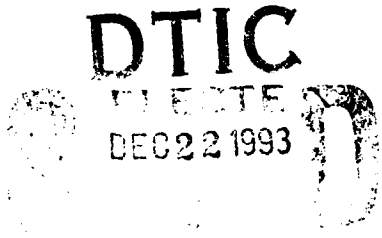
PROPOSED OBJECT ORIENTED PROGRAMMING (OOP) ENHANCEMENTS TO THE VERY HIGH SPEED INTEGRATED CIRCUITS (VHSIC) HARDWARE DESCRIPTION LANGUAGE (VHDL)

MICHAEL T. MILLS, LT COL

AUGUST 1993

FINAL REPORT FOR 05/04/92-08/04/93

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.



22pgs

SOLID STATE ELECTRONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-7331

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as in licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

Michael T. Mills

MICHAEL T. MILLS, LtCol, USAFR
Design Branch
Microelectronics Division

John W. Hines

JOHN W. HINES, Chief
Design Branch
Microelectronics Division

Stanley E. Wagner

STANLEY E. WAGNER, Chief
Microelectronics Division
Solid State Electronics Directorate

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/ELED, WPAFB, OH 45433-7331 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	AUG 1993	FINAL 05/04/92--08/04/93

4. TITLE AND SUBTITLE PROPOSED OBJECT ORIENTED PROGRAMMING (OOP) ENHANCEMENTS TO THE VERY HIGH SPEED INTEGRATED CIRCUITS (VHSIC) HARDWARE DESCRIPTION LANGUAGE (VHDL)

5. FUNDING NUMBERS
C
PE
PR
TA
WU

6. AUTHOR(S)
MICHAEL T. MILLS, LT COL

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SOLID STATE ELECTRONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7331

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES)
SOLID STATE ELECTRONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7331

10. SPONSORING MONITORING AGENCY REPORT NUMBER
WL-TR-93-5025

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report describes an integrated set of Object Oriented Programming (OOP) enhancements proposed for a future revision of the VHSIC Hardware Description Language (VHDL). It offers inheritance with extension for VHDL record, array and private type declarations, plus multiple inheritance with extension for entities and corresponding architecture bodies, and a class wide dispatching capability for all tagged types and tagged entities. If these three enhancements are accepted by the IEEE, VHDL based design automation tools can possess new abstract design capabilities for developing and enhancing electronic hardware. Current software languages with OOP capability increase productivity and reuse by enabling the design process to inherit and extend existing data structures and functionality. By selectively inheriting what already exists, the designer minimizes duplication. Functional capabilities and characteristics can be inherited and extended without affecting existing portions of a design.

14. SUBJECT TERMS	15. NUMBER OF PAGES
VHDL - IEEE 1076 - Design Language - Hardware Description Language, OOP - Object Oriented Programming	36
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

1. Introduction

This report describes a proposed Object Oriented Programming (OOP) set of enhancements to VHDL to be considered for the next or later revision of IEEE-Standard-1076 [1] or IEEE P1076 1992B [2]. First, it incorporates OOP features such as extended derived types into VHDL type declarations by borrowing some features from Ada 9X, the next revision to the Ada programming language. Second, it incorporates a new OOP technique, developed exclusively for this proposed enhancement, called derived entities. Third, it uses the class-wide type capability from Ada 9X by proposing a 'CLASS attribute for tagged types to support class-wide dispatching of operations. Certain types and all entities can be tagged which specifies that their characteristics and functional capabilities can be inherited (derived) and extended.

Record, array, and private type declarations with OOP provide the first proposed inheritance features. This allows new signals and variables to inherit characteristics (such as a data structure or bus width) from existing signals and variables. Any procedures or functions with parameters of the inherited type are also inherited. The designer can add new characteristics, functions, and procedures to the new signals and variables without disturbing the inherited signals or variables. These three OOP techniques will make VHDL a significantly more powerful abstract design language.

Derived entities provide the second proposed inheritance feature which uses syntax similar to Ada 9X but allow declarations and concurrent statements to be inherited from multiple entities. The designer can selectively modify, leave alone, or extend entities resulting in new capabilities created with minimal specification. If something other than what is inherited is desired, the designer specifies what is different from the inherited entities and their corresponding architectures. The designer still has to code complete concurrent statements, including whole process statements.

As a goal, this proposed set of language changes should stay upward compatible and minimize impact on existing VHDL tools. OOP normally incorporates inheritance to a class of objects with a set of operations associated with each class. The proposed OOP features for VHDL borrow derived types from Ada [3] and extensions of derived types from the proposed Ada 9X [4]. This proposal treats signals and variables as candidate VHDL objects to incorporate inheritance. Procedures and functions with signals and/or variables as its parameters or function returns serve as operations associated with these signals and variables. Since signal and variable declarations include the type construct (i.e., type_mark), adding extended derived types and a 'CLASS attribute for tagged types to VHDL will provide inheritance to

signals and, to a lesser extent, variables (including shared variables in VHDL-92B). Since the basic operations on signals and variables are assignment statements, in order to inherit such operations, the designer must encapsulate such assignment statements into procedures and (possibly) functions. Inheritance through derived types can only impact the type portion of a signal or variable. However, when used with other OOP features in this report, such inheritance can add significantly to the abstractness of VHDL and make VHDL designs more easily modifiable and reusable. By using the proposed extendable derived types for signals and variables and incorporating them into VHDL procedure and function parameters, operations such as signal or variable assignments can be inherited from existing subprograms into new subprograms and extended to provide new data structures. The designer can add new functional capabilities of inherited subprograms without affecting the existing subprograms.

2. *Benefits of Adding OOP Capability to VHDL*

As microelectronic designs continue to grow in complexity, design descriptions become overwhelmingly large. VHDL descriptions of this complexity could be simplified if new entities could inherit the characteristics and functionalities of existing entities and extend the characteristics and functionality without requiring any reanalyzing (or recompiling) of the old entities. Rather than duplicating a significant portion of existing entities into new entities, only differences between existing and new entities would need specification. For designs with the majority of its structure and behavior nearly regular (i.e., built with somewhat similar entities), such a VHDL enhancement could reduce complexity of VHDL code significantly.

If any existing entity, architecture, or signal were modified, the current VHDL would require that they be reanalyzed (i.e., recompiled). If new entities and architectures could inherit characteristics of existing ones and extend them without affecting their structure or behavior, designs could be expressed as differences from the existing entities and architectures which would no longer require reanalyzing. Both VHDL code size and design time would decrease significantly for most large designs. The appropriate Object Oriented features added to VHDL could make this a reality. Such VHDL changes would impact design, reuse, extendability and maintenance of the design description while adding significant design abstraction capability to the language. Existing designs are more adaptable for reuse if increased signal characteristics can be expressed in extendable derived types that leave these existing designs untouched. Such modifications reduce any degrading impact on the reliability of existing designs. The OOP capability described in this report is proposed for this purpose.

By adding OOP capability to the type declaration in VHDL, signals and variables can inherit characteristics from existing types. By modifying a signal declaration by changing its type, procedures which contain operations on these signals and variables can also be used with the new signals and variables. As a result, adapting procedures to modified signals with derived types becomes automatic.

With the following steps, the designer can develop procedures associated with signals and variables as building blocks to increase abstraction characteristics of VHDL design. (1.) Initially, develop a class of types and signals which have these types. Then, (2.) as these signals are more defined or specific, develop more specific procedures which are used by the more specific signals. When new signals are developed, which fall within the original class of signals, then procedures which were developed for that class can be used for the new signals without any modification. With extended derived types discussed below, large VHDL descriptions can be left intact without recompilation.

If this approach is implemented into a future VHDL revision and

corresponding design automation environments, productivity, reuse, reliability and maintainability of resulting microelectronic designs could potentially increase significantly. When new entities can be designed by specifying differences from one or more existing entities, productivity is likely to escalate. The more capabilities that new entities can inherit from existing entities, the greater reuse is implemented. Since existing entities are untouched and not reanalyzed, the original reliability remains high when adding new functionality and structure through derived entities, and architectures, and extendable derived types. Maintainability is enhanced by reducing duplicate specification and keeping capabilities of several entities localized to a few rather than many entities and corresponding architectures.

The late binding features of these proposed language enhancements provide a more dynamic language that opens new opportunities for implementations. Such language enhancements can significantly reduce impediments to faster, more dynamic design tools of the future approaching near instantaneous compilation and binding as the designer adds extended features to existing portions of his or her design.

The proposed OOP features in this report require the following reserved words to be added to the VHDL standard:

new
is new
with record
with array
with private
tagged (followed by record, array, private, or entity)
(also, tagged is in the phrase **is tagged**.)
is derived
with null
with delete

Since most of these reserved words are only allowed in restricted locations within the language and are usually accompanied with another reserved word, chances of these new reserved words causing the proposed OOP changes to be nonupward compatible is expected to be zero.

3. OOP in Small Talk, C++, and the Proposed Ada 9X

Determining an appropriate OOP approach for VHDL began by examining OOP implementations of existing software languages such as Small Talk, C++ and the current draft revision of Ada, called Ada 9X. After looking at these OOP approaches, considering advantages and disadvantages of each, and considering appropriateness with existing VHDL features, I selected Ada 9X as the most appropriate baseline from which to develop OOP enhancements to VHDL. As a member of the Ada 9X Distinguished Reviewer Team, I had up to the minute access on the current state of Ada 9X and was most familiar of its OOP approach compared to the other languages.

Small Talk's approach to OOP is to allow everything to be inherited. This approac: has a tremendous implementation cost which is unacceptable for large and complex designs normally implemented by VHDL.

The C++ approach to OOP, although well known by a large software community, had several drawbacks when compared to Ada 9X. [3] contains a technical report on Ada 9X vs. C++. Due to its evolution from C, a language with unsafe features and with little to offer from which to build abstractions, C++ had to implement new features to do what already existed in the Ada language, such as information hiding. The unsafe characteristics of C, especially for large complex programs, is still an irritant for C++.

Ada 9X was the best approach because of several reasons. VHDL was originally designed using Ada as a baseline. As a result, VHDL became very Ada like. The Ada 9X OOP approach works well with large, complex programs. It also builds from features already in the Ada language.

The OOP approach, proposed for VHDL in this report, borrows some of the OOP features from Ada 9X, such as derived and tagged types. It also contains some unique OOP features, which are Ada 9X like in syntax, but were developed exclusively for VHDL. These are derived entities and architectures. This feature was necessary to apply OOP where it could have its biggest impact, on the VHDL entity, the key to VHDL abstraction. Ada 9X derived types used by this approach apply only to the type portion of declared signals and variables. The VHDL derived types are proposed for record, array, and private types.

Another proposed OOP feature borrows class-wide types from Ada 9X which provides polymorphic dispatching of operations to class-wide types which are tagged. A 'CLASS attribute is proposed to provide classes of tagged types and their derivitives. This will provide a more complete abstraction capability to VHDL when integrated with the other two proposed OOP enhancements.

4. *The Proposed OOP Enhancements to VHDL Types*

By adding derived types (from Ada 83) with extended parts (like those proposed for Ada 9X), VHDL can incorporate Object Oriented Programming capability within its data types.

In current VHDL, signals can reside in Architecture Bodies, Blocks, Packages, Entities, and Parameters of Subprograms. Variables are restricted to Processes and Subprograms. This proposed OOP approach treats VHDL signals and variables as objects. Signal Assignments and Variable Assignments, whether concurrent or sequential, are operations on signals and variables. Procedure calls can be overloaded based on their parameter types. Signals and variables are declared with types and, therefore, can be used as formal parameters of subprograms to differentiate which overloaded subprogram can be used as one of its operations.

The following changes are proposed to VHDL types in order to add object oriented programming with inheritance to VHDL signals and variables and the procedures and functions associated with them.

1. Add a derived type to the type definition as follows:

```
type_definition ::=
   scalar_type_definition
   | composite_type_definition
   | access_type_definition
   | file_type_definition
   | derived_type_definition

derived_type_definition ::=
   new subtype_indication [type_extension_part]

type_extension_part ::=
   record_extension_part
   | array_extension_part
   | private_extension_part

record_extension_part ::=
   with record
      element_declaration {,element_declaration}
   end record

array_extension_part ::=
   with array
      [ modified_index_constraint ]
      [ of modified_element_subtype_indication ]
   end array

private_extension_part ::=
   type identifier is new
```

*ansister*_subtype_indication **with private** ;

The reserved word **new** is added to the language to designate that the type is derived from an existing type. The type_extension_part is only allowed for tagged types discussed below.

2. Add tagged types to the language by adding the reserved word **tagged** to the array and record type definition. Adding this reserved word to the composite_type_definition provides extended inheritance for record and array types.

```
composite_type_definition ::=
   [tagged] array_type_definition
   | [tagged] record_type_definition
```

The tagged feature can provide the OOP polymorphism (i.e., dynamic binding) to the language. In Ada 9X, this feature includes dynamic (run-time) polymorphism. Tagged types provide dynamic binding and dispatching of inherited features at run time. The implementation may provide static checking at compile (or analysis) time for some OOP binding. However, when essential OOP binding information is only known at simulation time, dynamic binding capability is required.

The proposed Ada 9X includes only record and private types as tagged [4]. An early version of the Ada 9X design proposed array types as tagged but was later dropped to keep the language complexity and implementation costs to a minimum. Since array types in VHDL are used heavily, this proposal currently includes tagged and derived types for arrays as well as records. If implementation costs are too high for VHDL as well, then that part of the proposed enhancement could be dropped without impacting the rest of this OOP enhancement to VHDL.

Assuming private types either survive the IEEE balloting process for the VHDL-92 standard or appear in a future revision, this OOP proposed enhancement also includes tagged and derived types for private types (as in Ada 9X) for added abstraction capability.

3. This proposal adds a class concept for tagged types to the language. Ada 9X contains this feature by adding a 'CLASS attribute for class-wide types. An Ada 9X class specifies a base type plus all its derivatives which can have common characteristics and operations. Class-wide types provide the internal dispatching capability of tagged types for late binding.

Record, Array, and Private declarations with tags added:

```
record_type_declaration ::=
      [ tagged ] record
            element_declaration
            { element_declaration }
      end record ;
```

```
constrained_array_definition ::=
      [ tagged ] array index_constraint of
            element_subtype_indication

unconstrained_array_definition ::=
      [ tagged ] array ( index_subtype_definition
           index_subtype_definition } ) of
                 element_subtype_indication

private_type_declaration ::=
      type identifier is [ tagged ]
           private [ contractual-details ] ;
```

5. *An Example of Using Tagged and Derived Types in VHDL*

```
package ABSTRACT is
     type BUSS_SMALL is tagged array (0 to 15) of BOOLEAN;

entity SOMETHING is
     :
end ...;

architecture DATA_FLOW of SOMETHING is
     signal SMALL_SIG : BUS_SMALL;

begin
     :
     SMALL_SIG <= ...
     :
end ...;

use package ABSTRACT;
package MORE_SPECIFIC is
     type BUS_LARGE is derived BUS_SMALL
                          with array (16 to 31) of ... ;
                                    -- This type inherits.
          :
     signal LARGER_SIG : BUS_LARGE;  -- New signal
                          -- indirectly inherits from old
                          -- signal through derived type
                          -- inheritance.
          :
end ...;
```

6. *OOP Enhancements to VHDL Entities and Architectures*

Add a new capability for new entities to be derived from existing entities and to inherit all characteristics from the existing entities and their corresponding architectures. Add an optional tagged indicator (a new reserved word **tagged**) to the original entity declaration to specify inheritance at simulation time.

```
entity_declaration ::=
  [tagged] entity identifier is
             entity_header
             entity_declarative_part
  [ begin
             entity_statement_part ]
  end entity [entity_simple_name];
```

Derived Entities:

Add a new VHDL construct called a derived entity which references any existing entity declaration and specifies any modifications. The derived entity inherits all the functional, structural, and declared object (signal, variable, and constant) characteristics of the entity it references. The entity_extension_part of the derived entity contains any added or modified parts that are different from the existing (or referenced) entity. This gives the designer a capability of creating new entities that are similar to existing entities by only specifying changes. None of the existing entities are corrupted and, therefore, should not have to be reanalyzed when analyzing and linking the derived entities. The new key word **tagged** is added to existing entities to specify inheritance at simulation time (often called polymorphism). This feature tells the tool environment to do all that is necessary to prepare the entity for inheritance at simulation time. Only specify one *inherited_entity*_identifier unless inheritance is desired from more than one entity (referred to as multiple inheritance). When multiple inherited entities are specified, the first one listed is the default to resolve conflicting declarations and concurrent statements from different inherited entities. To selectively deviate from the default, explicitly write the declaration of the concurrent statement. Note: Derived entities can also be tagged so that their characteristics and functionality can be inherited by entities derived and extended.

```
derived_entity_declaration ::=
     [tagged] entity identifier  is new
        inherited_entity_identifier
          { is new inherited_entity_identifier }
                              [ entity_extension_part  ]


entity_extension_part ::=
     with entity
          entity_extensions_list
```

```
        end [entity] [entity_simple_name];


    entity_extensions_list ::=
            [entity_header_changes]
            [entity_declarative_part_changes]
        [ begin
            entity_statement_part_changes]
```

Include in the entity_extensions_list all added or modified declarations, statements, generics, or ports. To copy items from the derived entity without adding extensions specify with the reserved words "**with null**" at the beginning of the item.

```
entity_header_changes ::=
      [ [with null] formal_generic_clause ]
      [ [with null]  formal_port_clause ]

entity_declarative_part_changes ::=
    { [with null]  entity_declarative_item }

entity_statement_part_changes ::=
      { [with null]  entity_statement }
```

The above new VHDL features allow the designer to specify differences (additions or modifications) between a new derived entity and the existing entity it references without having to rewrite any significant portion of the existing entity.

Note: An earlier design of entity_statement_part_changes revealed that diminishing returns might be reached if too many changes are specified. In this case, single inheritance would be more appropriate than multiple inheritance. For some entities and architectures, no inheritance would be more appropriate. This approach provides controlled single or multiple inheritance either statically (by the analyzer) or dynamically (by the simulator). Derived entities and architectures can be inherited by other additional derived entities and architectures. As a result, abstraction can be implemented extensively in a design.

For declarations and concurrent statements within nested block statements, the block structure is specified by rewriting the block statements. Respecifying the block, or nested block, structure is normally a minimal effort and avoids added syntax complexity.

Derived Architectures:

The following new VHDL features perform similar functions as above for derived architectures. This allows designers to specify differences (additions, modifications, and deletions) between an architecture body of a new derived entity and the referenced architecture of an existing entity. For simulation time inheritance, only the entity of the referenced architecture

is **tagged**. Although changes are specified within the derived architecture, all inheritance is accomplished through the entity, the chief VHDL building block for abstraction.

```
derived_architecture_declaration ::=
    architecture identifier of derived_entity_name
                    is [tagged]
                  new inherited_architecture_name
                { is new inherited_architecture_name }
                        [architecture_extension_part]
              end [architecture][architecture_simple_name];
```

The architecture_extension_part includes items to be added or modified. Items to be deleted are designated "**without**".

```
architecture_extension_part ::=
        [architecture_declarative_part_changes]
    [begin
        [architecture_statement_part_changes]

architecture_declarative_part_changes ::=
      { [without] block_declarative_item }

architecture_statement_part_changes ::=
      { [without] concurrent_statement }
```

Both block_declarative_items and concurrent statements within the derived architecture can repeate exactly in the architecture of the inherited entity, modified (using the name of the item or statement and changing what is desired), added as a new item or statement, or deleted using a **with delete** at its beginning. Existing items or statements which are not changed do not need to be repeated. The analyzer will include them as part of the new architecture. To reference nested parts of an architecture, repeat the surrounding nested constructs and anything else necessary to resolve any conflicting names, etc.

Note: Since derived entities can be tagged, there is no reason to have a tag for the corresponding derived architecture (for further inheritance).

Note: An earlier design of architecture_statement_part_changes provided the capability to selectively change the default inherited entity for each (or a group of) block_declarative_items or concurrent_statements. However, this feature complicated the syntax. Therefore, in order to avoid complexity, the syntax was changed so that the designer would rewrite or modify the declaration or concurrent statement that deviates from the architecture of the defaulted inherited entity. Unique declarations and concurrent statements are still inherited from their respective architectures (or entities), whether it is the default or not. The earlier draft syntax for architecture_declarative_part_changes and architecture_statement_part_changes was

```
 {[of inherited_entity_identifier]
          {[with null] entity_statement }}
```

The "*inherited_entity*_identifier" changed the default for a single or a group of declarations or statements. The "*inherited_entity*_identifier" rather than the "*inherited_architecture*_name" was used as a selection mechanism since multiple inheritance consists of inheriting from architectures of corresponding multiple entities rather than multiple architectures.

Note: The number of changes made within a derived entity or architecture should be limited to a reasonable number to maintain readability of the VHDL source code. If the number of changes exceeds this, then best engineering practice dictates that the new entity and its architectures be written as actual entities and architectures that are not derived. The purpose of derived entities and its corresponding derived architectures and inheritance is to reuse what exists, adapt these entities, and incorporate a limited number of changes without having to create behaviors, structures, and objects that already exist in the inherited entities or their respective architectures.

7. *An Example of Using Derived Entities and Architectures*

Existing entity and architecture body:

```
tagged entity Full_Adder is
     port (X, Y, Cin: in Bit, Count, Sum: out Bit);
end Full_Adder;

architecture Data_Flow of Full_Adder is
     signal A, B: Bit;
begin
     A <= X xor Y;
     B <= B and Cin;
     Sum <= A xor Cin;
     Cout <= B or (X and Y);
end DataFlow;
```

Derived entity and architecture body (only specifying changes):

```
entity Modified_Adder is new Full_Adder
     with entity
     generic (new_stuff : integer) -- This is added.

-- This port is unchanged, therefore unspecified.
begin
     New_Object : BIT;    -- only extensions
end entity Modified_Adder;
                -- Statements from inherited
                     -- entity are implicitly
                          -- coppied here.


architecture Worp_Flow of Modified_Adder is
     new Data_Flow
begin
     with null
      Sum <= A xor Cin;
     Intermediate <= A xor Cin;  -- added
     Sum < Intermediate and A;   -- modified
          -- Note: The remaining declarations and
          --       statements remain unchanged.
end architecture Worp_Flow;
```

Note: More complicated examples would show more obvious reuse and design time/cost savings. Note that the original entities and architectures are untouched and, therefore, remain error free. Since the changes are minimized and the bulk of design is localized in existing entities and architectures, the resulting code is highly reliable and maintainable compared to code that duplicates a significant portion of predesigned code.

8. Classes of Types

This proposed VHDL enhancement provides dispatching on pimitive operations of tagged types. It borrows this from Ada 9X.

Ada 9X provides the following:

"Ada 9X provides dispatching on the primitive operations of tagged types. When a primitive operation of a tagged type is called with an actual parameter of a class-wide type, the appropriate implementation is chosen based on the tag of the actual value. This choice is made at run time and represents the essence of dynamic polymorphism. ... In some cases, the tag can be determined at compile time." [4]

9. *Rationale for adding OOP to VHDL*

Since concurrent procedure calls can be called from architecture bodies, entities (if passive), and blocks, and since nonconcurrent procedure calls can be called from processes and subprogram bodies (functions and procedures), then procedures are good candidates for primitive operations. Since procedures (and functions) can be overloaded and, therefore, selected by their parameter and return types, these operations can be associated with objects of the same type. Formal parameters and returns of procedures and functions can be constants, signals, or variables. Since constants cannot be changed, signals and variables are the remaining objects which can be operated on by procedures. Since the scope of which variables are visible is limited to processes and subprograms, their application as objects for inheritance is limited. However, they may have some value for abstraction for objects confined within its scope. Global variables with their increased scope offer additional objects for inheritance but are only an added convenience. None of the OOP features in this report depend on global variables. The remaining and most likely objects associated with procedures and candidates for OOP are signals. Signals can be operated on from within architecture bodies, blocks, processes, and subprogram bodies. The common operation which affects signals is the signal assignment statement. Since procedure calls can be located anywhere signal assignment statements are present, one or more signal assignment statements can be placed in procedures (subprogram body declarations). Then, such procedures can be overloaded and associated with signals that are formal parameters of a given type. If the types of such signals contain inheritance capabilities, then adapting processes containing such procedure calls to modified types of signals is made easier as a result of increased abstraction within VHDL.

As procedures accumulate in a design, they can be placed in packages and imported from a library. These procedures provide such operations reusable. Signals with inheritance capability can be declared abstractly and be associated with a set of common procedures as its operations. As a design progresses, new signals with types derived from more abstract types can use the original procedures plus new ones specific for the new (derived) signals. Characteristics of these signals can be extended in a polymorphic manner by designating its base type as tagged. Thus, stepwise refinement in signal definition with its associated operations (signal assignment statements embedded in its associated procedures) offers a method of providing greater design abstraction within VHDL design descriptions and simulations.

The following example illustrates the above concept.

```
-- Declaring abstract types:

 type A is tagged array (A1 : BOOLEAN) of BOOLEAN;
```

```
type B is new A with ......
type C is tagged record
                      ......
                      ......
                   end record;
signal X : A;
signal Y : B;
signal Z : C;

procedure M (X:A)                    procedure M (Y:B)
begin;                                 begin;
   A_1 <= transport X;                    M_1 <= transport Y;
   A_2 <= ...;                            M_2 <= ...;
   ...                                    ...
end;                                   end;

process
   M (X:A);
   M (Y:B);
   ...
end;
```

10. References

1. The Institute of Electrical and Electronic Engineers, Inc., *IEEE Standard VHDL Language Reference Manual*. IEEE Std 1076-1987.

2. The Institute of Electrical and Electronic Engineers, Inc., *IEEE Standard VHDL Language Reference Manual - Draft* (The 1992 Draft Revision currently being balloted). IEEE Proposed Std VHDL 1992B Draft.

3. American National Standards Institute, *Reference Manual for the Ada Programming Language*. ANSI/MIL-Std-1815a edition, 1983.

4. Ada 9X Mapping/Revision Team, Intermetrics, Inc., *Annotated Ada 9X Reference Manual - Draft*, Version 2.0, 29 March 1993.